Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jure Jesenšek

Prenos orodja SYCL-GTX na operacijski sistem Linux in koprocesor Xeon Phi

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Orodje SYCL-GTX je odprtokodna implementacija standarda SYCL. V osnovi je bila razvita na operacijskem sistemu MS Windows in podpira uporabo grafičnih procesorskih enot, na katerih je mogoče uporabljati OpenCL. Prenesite orodje SYCL-GTX na operacijski sistem Linux in zagotovite podporo za koprocesor Xeon Phi. Izmerite hitrost delovanja SYCL-GTX v novem okolju, hkrati pa preverite, ali so bile originalne meritve hitrosti orodja SYCL-GTX zanesljive.

Zahvaljujem se mentorju, doc. dr. Boštjanu Slivniku za pomoč, usmerjanje in potrpežljivost pri nastajanju diplomskega dela.

Družini, prijateljem in sošolcem se zahvaljujem za izkazano podporo, zaupanje, potrpežljivost in pomoč pri pisanju diplomskega dela, kot tudi tekom celotnega študija.

Prav tako se zahvaljujem asistentu Davorju Slugi za zagotavljanje tehnične podpore pri izdelavi dela.

Družini in prijateljem.

Kazalo

Seznam uporabljenih kratic

API	Application programming interface	Aplikacijski programski vmesnik
CPE	Central processing unit	Centralna procesna enota
CUDA	Compute unified device architecture	Arhitektura enotnih računskih naprav
DSP	Digital signal processor	Signalni procesor
FPGA	Field-programmable gate array	Programabilno logično vezje
GPE	Graphical processing unit	Grafična procesna enota
GPGPU	General-purpose computing on GPU	GPE namenjena splošnemu računanju
ICC	Intel C Compiler	Intelov prevajalnik jezika C
JIT	Just-in-time	Sprotno prevajanje
MIC	Many Integrated Core	Veliko integriranih jeder
RTTI	Runtime Type Information	Informacije o tipih med izvajanjem

Povzetek

Naslov: Prenos orodja SYCL-GTX na operacijski sistem Linux in koprocesor Xeon Phi

Avtor: Jure Jesenšek

V zadnjem času je poleg računanja na večjedrnih centralnih procesnih enotah (CPE), postalo pogosto tudi računanje na vedno zmogljivejših grafičnih procesnih enotah (GPE). Tak način računanja, ki mu pravimo heterogeno programiranje, pa prinaša probleme pri prenosljivosti programske opreme. Ena izmed rešitev je uporaba OpenCL, ki poveča prenosljivost napisane programske kode, ampak ni najbolj preprost za uporabo. Za rešitev teh problemov je bil izdan nov soroden standard SYCL, ki prinaša mnoge poenostavitve v primerjavi z OpenCL.

Naloga opiše standarda OpenCL in SYCL ter ju primerja med seboj. Predstavi odprtokodno implementacijo standarda z imenom sycl-gtx in nato nadaljuje z opisom rezultatov testiranj na različnih tipih strojne opreme. Testiranje se izvaja na CPE in GPE različnih zmogljivosti, kot tudi na Intelovem koprocesorju Xeon Phi. Naloga se konča s komentarjem in primerjavo dobljenih rezultatov testiranj z že obstoječimi testiranji, ki jih je izvedel avtor sycl-gtx.

Ključne besede: SYCL, OpenCL, heterogeno programiranje, vzporedno programiranje.

Abstract

Title: Migrating SYCL-GTX to Linux and Xeon Phi coprocessor

Author: Jure Jesenšek

Besides the use of multi-core central processing units (CPUs) there has been an increase in use of evermore powerful graphics processing units (GPUs) for the purpose of parallel computing. But this kind of simultaneous computing on different types of processing units, called heterogeneous programming, brings on a new set of challenges, especially those concerning software portability. One of the more commonly used solutions to this problem is OpenCL framework, which is aimed at maximising portability across platforms, but is not the easiest to use. To solve this and other issues, a new standard, named SYCL, that aims to simplify heterogenous programming was released.

This thesis describes and compares the OpenCL and SYCL standards. It introduces an open-source implementation of the SYCL standard called sycl-gtx and continues with the presentation of test results on different kinds of hardware, from CPUs and GPUs with different levels of performance, to the Intel's manycore processor Xeon Phi. Finally, the results are compared to those obtained by the original author of sycl-gtx and a conclusion is presented.

Keywords: SYCL, OpenCL, heterogeneous programming, parallel programming.

Poglavje 1

Uvod

Zmogljivost računalniške strojne opreme narašča praktično od začetka razvoja računalništva. Znani Moorov zakon napoveduje pohitritev centralnih procesnih enot (CPE) za faktor 2 na vsaki dve leti - kljub upočasnitvi naraščanja zmogljivosti v zadnjih letih, pa zakon še vedno drži. V preteklosti je ta pohitritev prišla s strani povečevanja takta procesorja, vendar pa je ta način pohitritve dosegel točko, kjer obstoječa tehnologija izdelovanja čipov ne omogoča več višjih frekvenc. Zato se je razvoj CPE usmeril v povečevanje števila jeder, ki lahko vzporedno izvajajo programsko kodo.

V zadnjem času so se za primerno platformo, na kateri je mogoče vzporedno izvajati računske probleme, izkazale tudi grafične procesne enote (GPE), ki so se jim kasneje pridružile še GPE namenjene računanju (angl. *General-purpose computing on graphics processing units - GPGPU*). Slednje so specializacija prvotnih GPE, ki z nadgradnjami, kot so izboljšan cevovod, podpora za vektorizacijo (računske operacije na več številih hkrati) in podpora številom zapisanim z dvojno natančnostjo (angl. *double precision floating point number - double*), ciljajo na uporabnike, ki uporabljajo GPE za računanje, in pogosto sploh ne omogočajo priklopa monitorja.

Razvoj aplikacij, ki vršijo paralelne izračune na GPE, pa poleg problemov, ki jih prinaša razvoj novih, oziroma selitev obstoječih algoritmov, prinaša tudi dodatne probleme pri prenosljivosti zaradi razlik v arhitekturi GPE

različnih proizvajalcev. Eden izmed standardov, ki je nastal z namenom premostitve teh razlik med GPE, je programsko ogrodje (angl. *framework*) OpenCL [10], nad katerim bdi skupina Khronos [13].

Poleg precejšnjega števila dobrih lastnosti, pa ima standard OpenCL tudi nekaj pomanjkljivosti. Že samo programiranje OpenCL programov ni najbolj preprosto, saj mora programer sam skrbeti za precejšnje število nizko nivojskih podrobnosti, kot so dodeljevanje in sproščanje pomnilnika, inicializacija potrebnih struktur in podobno. Prav tako programski jezik standarda OpenCL temelji na programskem jeziku C, ki je sicer učinkovit, vendar ne vsebuje konstruktov višjih programskih jezikov, ki poenostavljajo programiranje. Zaradi tega je marca 2014 skupina Khronos predstavila osnutek standarda SYCL [12], ki se naslanja na OpenCL in naj bi poenostavil heterogeno programiranje, tokrat z uporabo programskega jezika C++.

Kljub obetavnosti standarda SYCL pa kar nekaj časa po njegovi objavi ni bilo mogoče zaslediti konkretnejše implementacije. Podjetje Codeplay Software, ki je eno izmed glavnih sodelujočih pri sestavljanju standarda SYCL, ima sicer svojo implementacijo [3], vendar pa ni prosto dostopna. Obstaja tudi odprtokodna implementacija triSYCL [15], katere razvoj pa očitno stagnira.

Zato je leta 2016 na Fakulteti za računalništvo in informatiko v Ljubljani nastal predlog za razvoj lastne odprtokodne implementacije standarda SYCL, ki ga je realiziral Peter Žužek v svoji magistrski nalogi [22] z naslovom *„Implementacija knjižnice SYCL za heterogeno računanje“*. Zaradi lažjega in hitrejšega razvoja se ni odločil razviti posebnega prevajalnika, temveč je SYCL implementiral kot C++11 knjižnico, ki vrši prevajanje v času izvajanja (angl. *Just-In-Time - JIT*). Njegova implementacija standarda SYCL, z imenom sycl-gtx [21] in licenco MIT, je tudi prosto dostopna preko spleta.

Eden izmed večjih problemov implementacije sycl-gtx je bila v tem, da je bila razvita na platformi Microsoft Windows in na začetku ni bila prenosljiva na ostale platforme oziroma operacijske sisteme. Prav tako je bil sycl-gtx napisan in testiran zgolj na grafičnih procesnih enotah (GPE), nas pa je

zanimalo, ali bi se ga dalo uporabiti tudi na Intelovi družini koprocesorjev Xeon Phi [7] tehnologije MIC, ki so na razpolagi na fakulteti. Prav tako so nas zanimale razlike v hitrostih izvajanja na različnih platformah, od navadnih CPE v stacionarnih osebnih računalnikih in prenosnikih, preko strežniških CPE Intel Xeon [6], pa vse do GPE namenjenim računanju Nvidia Tesla K20 [9] in prej omenjenega koprocesorja Xeon Phi.

V 2. poglavju naloga opiše standarda OpenCL in SYCL ter ju primerja med seboj. Poglavje 3 predstavi glavni del naloge - prenos obstoječe implementacije SYCLa (sycl-gtx) na operacijski sistem Linux in koprocesor Xeon Phi. Predstavimo sycl-gtx in ciljno programsko opremo. V 4. poglavju opišemo še preostalo strojno opremo, na kateri smo testirali ter predstavimo meritve rezultatov in ugotovitve. Poglavje 5 strne sklepne ugotovitve in rezultate ter zapiše nekaj stavkov o prihodnosti implementacije sycl-gtx.

Poglavje 2

Heterogeno programiranje z OpenCL in SYCL

2.1 OpenCL

OpenCL (*Open Computing Language*) [10] je odprt in prosto dostopen standard za paralelno programiranje heterogenih sistemov - računalniških sistemov, na katerih je mogoče izvajati računske operacije na različnih vrstah procesorjev. Standard podpirajo praktično vsi veliki igralci na področju proizvodnje računalniške strojne opreme. Zaradi te široke podpore lahko OpenCL izvajamo na širokem spektru naprav - na centralnih procesnih enotah (CPE), grafičnih procesnih enotah (GPE), signalnih procesorjih (angl. *Digital Signal Processor - DSP*), programabilnih logičnih vezjih (angl. *Field-Programmable Gate Array - FPGA*) in tako naprej. OpenCL omogoča programerjem, da napišejo en sam ciljni program, ki pa je prenosljiv med različnimi izvajalnimi enotami (napravami) v heterogenem sistemu. Problem, ki ga programer želi računalniško rešiti, se lahko tako izvaja na napravi, za katero programer meni, da je najbolj primerna za reševanje problema. CPE pa usklajuje samo izvajanje celotnega programa.

OpenCL je pričelo razvijati podjetje Apple [1] in je avgusta 2009, z izdajo operacijskega sistema Mac OS X Snow Leopard, objavilo tudi prvo

implementacijo standarda. Skrb za OpenCL je že pred tem prevzela skupina Khronos [13], ki je novembra 2008 izdala specifikacijo OpenCL 1.0. Podjetje AMD se je posledično odločilo, da bo namesto nadaljevanja razvoja lastnega vmesnika za računanje z imenom *Close to Metal*, raje podprlo OpenCL. Nvidia pa poleg podpore za OpenCL, vzporedno razvija tudi lasten vmesnik za računanje *CUDA* (*Compute Unified Sevice Architecture*) [8], ki pa ga je za razliko od OpenCL mogoče uporabljati le na Nvidijinih GPE.

Z vidika standarda OpenCL je računalnik sestavljen iz dveh glavnih razredov naprav: gostiteljske naprave, ki skrbi za pripravo in nastavitve izvajanja, ter ciljne naprave, ki izvaja del kode, ki jo želimo paralelizirati z OpenCL. Kodo, ki se izvede na ciljni napravi OpenCL, imenuje ščepec (angl. *kernel*), ki ga programer poda preko ločene datoteke (ponavadi s končnico *.cl*), ali pa preko niza znakov (angl. *string* oziroma *char array*). To je tudi ena izmed nevšečnosti standarda OpenCL, saj zahteva ločevanje ščepca od preostale gostiteljske kode, le-tega pa je tudi potrebno pisati v posebni izpeljanki jezika C, ki se imenuje OpenCL C.

Standard sicer sestavljajo programski vmesnik OpenCL API, prej omenjen programski jezik OpenCL C, ter gonilniki in knjižnice, ki omogočajo izvajanje na različnih napravah. Programski jezik OpenCL C temelji na ISO standardu C99, vendar vsebuje dodatke za paralelno programiranje.

V času pisanja te naloge je bila najnovejša verzija OpenCL 2.2, ki kot prva podpira uporabo jezika C++ v ščepcih. V nalogi se bomo fokusirali na prvo široko uporabljano verzijo OpenCL, in sicer verzijo 1.2.

Sledi primer potrebnih korakov za zagon OpenCL aplikacije s preprostim ščepcem.

2.1.1 Osnovni algoritem

V izseku programa 2.1 je prikazan preprost algoritem za seštevanje vektorjev celih števil, ki ga bomo v nadaljevanju paralelizirali, najprej z OpenCL, nato pa še s SYCL.

```
1 int a[N], b[N]; // vektorja napolnjena s števili
```

```
2 int result[N];
3 for(int i = 0; i < N; i++)
4 {
5   result[i] = a[i] + b[i];
6 }
```

Izsek programa 2.1: Seštevanje vektorjev.

2.1.2 OpenCL ščepec

V izseku 2.2 je predstavljen predelan osnovni algoritem v ščepec, ki se lahko prevede in izvede z OpenCL.

```
1  __kernel void add(__global int *a, __global int *b,
2                    __global int *result, int N)
3  {
4    int i = get_global_id(0);
5    // preprečimo dostope izven polja
6    if (i < N)
7    {
8      result[i] = a[i] + b[i];
9    }
10 }
```

Izsek programa 2.2: OpenCL ščepec.

OpenCL ščepce programer piše podobno kot ostale funkcije v jeziku C, vendar z nekaterimi omejitvami oziroma dodatki. Funkcijo, ki predstavlja ščepec, označimo z določilom `__kernel` in ne sme vračati vrednosti (mora biti tipa `void`). Ščepec pa lahko kliče ostale ščepce, ki morajo biti prav tako označeni s `__kernel`. Na kratko bi našteli najbolj opazne omejitve, med katerimi standard ne dovoljuje:

- rekurzivnih klicev v ščepcih,
- kazalcev na funkcije (angl. *function pointer*),
- polj in struktur spremenljive dolžine (angl. *variable-length arrays*).

Če preko argumentov podajamo kazalce (angl. *pointer*), jih moramo označiti z določili `__constant`, `__local` ali `__global`, in sicer glede na to, ali kažejo na spomin kjer so konstante, ali na lokalni oziroma globalni spomin.

Ščepcu v izseku 2.2 preko argumentov podamo kazalce na oba vhodna vektorja števil (predstavljena kot polji - angl. *array*) ter na vektor, kamor se zapisujejo rezultati seštevanj. Ker se ti vektorji nahajajo v glavnem pomnilniku, je potrebno kazalcem dodati določilo `__global`. Parameter `int N` uporabimo pri določitvi meje izvajanja kode. S klicem funkcije `get_global_id(0)` ščepec dobi celoštevilsko vrednost, ki predstavlja položaj posameznega jedra v napravi. Tako se določi na katerem položaju v zaporedju števil mora jedro vršiti seštevanje, saj vsako jedro sešteva le enkrat (v primeru da je število jeder večje od velikosti vektorja). Ker se kopija ščepca izvede na vsakem jedru v GPE, bi lahko posledično v primeru, ko je število jeder v GPE različno od dolžine vektorja dostopali do naslovov v spominu, ki so izven rezerviranega območja spomina (angl. *buffer overflow*). Rezultat seštevanja se zapiše v polje `result`, ki ga CPE po koncu izvajanja prenese v glavni pomnilnik.

2.1.3 Gostiteljska OpenCL koda

Poleg samega ščepca je za izvajanje OpenCL aplikacije potrebno napisati tudi kodo, ki skrbi za vse preostale vidike OpenCL aplikacije. V izseku 2.3 je opisana poenostavljena koda, ki skrbi za pripravo naprave in ostalega okolja za izvedbo OpenCL ščepca na želeni napravi. V osnovi je izvajanje OpenCL aplikacije z vidika gostiteljske naprave sestavljeno iz štirih korakov:

1. prenos podatkov na ciljno napravo,
2. podajanje zahteve za izračun,

3. računanje,

4. prenos podatkov nazaj v glavni pomnilnik.

Izsek 2.3 prikaže kako v grobem izgleda koda za zagon našega ščepca iz izseka 2.2 na ciljni napravi. V primeru je izpuščeno preverjanje napak pri izvajanju.

```
1  int a[N], b[N], result[N];
2  cl_int returnValue;
3
4  // 1. platforma
5  cl_uint numOfPlatforms;
6  clGetPlatformIDs(0, nullptr, &numOfPlatforms);
7  std::vector <cl_platform_id>
8      platforms(numOfPlatforms);
9  clGetPlatformIDs(
10     numOfPlatforms, platforms.data(), nullptr);
11 cl_platform_id platform = platforms[0];
12
13 // 1. naprava 1. platforme
14 cl_uint numOfDevices;
15 clGetDeviceIDs(
16     platform, CL_DEVICE_TYPE_GPU,
17     0, nullptr, &numOfDevices);
18 std::vector <cl_device_id> devices(numOfDevices);
19 clGetDeviceIDs(
20     platform, CL_DEVICE_TYPE_GPU,
21     numOfDevices, devices.data(), nullptr);
22 cl_device_id device = devices[0];
23
24 // kontekst
25 cl_context context = clCreateContext(
```

```
26      nullptr, 1, &device, nullptr,
27      nullptr, &returnValue);
28
29  // ukazna vrsta
30  cl_command_queue command_queue =
31      clCreateCommandQueue(
32          context, device, 0, &returnValue);
33
34  // rezervira in nastavi prostor
35  // v pomnilniku na napravi
36  cl_mem bufferA = clCreateBuffer(
37      context,
38      CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
39      N * sizeof(int), (void*) a, &returnValue);
40  cl_mem bufferB = clCreateBuffer(
41      context,
42      CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
43      N * sizeof(int), (void*) b, &returnValue);
44  cl_mem bufferResult = clCreateBuffer(
45      context,
46      CL_MEM_WRITE_ONLY,
47      N * sizeof(int), nullptr, &returnValue);
48
49  // prebere ščepec iz datoteke
50  std::string kernelSource = read("kernelExample.cl");
51  const char* kernelPtr = kernelSource.c_str();
52
53  // prevede ščepec
54  cl_program program = clCreateProgramWithSource(
55      context, 1, &kernelPtr, nullptr, &returnValue);
56  clBuildProgram(
```

```
57      program, 1, &device, nullptr, nullptr, nullptr);
58
59  // pripravi ščepec in argumente za ščepec
60  cl_kernel kernel = clCreateKernel(
61      program, "add", &returnValue);
62  clSetKernelArg(
63      kernel, 0, sizeof(cl_mem), (void*) &bufferA);
64  clSetKernelArg(
65      kernel, 1, sizeof(cl_mem), (void*) &bufferB);
66  clSetKernelArg(
67      kernel, 2, sizeof(cl_mem),
68      (void*) &bufferResult);
69  clSetKernelArg(
70      kernel, 3, sizeof(cl_int), (void*) &N);
71  size_t globalItemSize[] = {N};
72
73  // požene ščepec
74  clEnqueueNDRangeKernel(
75      command_queue, kernel, 1, nullptr,
76      globalItemSize, nullptr, 0, nullptr, nullptr);
77
78  // počaka da se izvajanje zaključi
79  clFinish(command_queue);
80
81  // prebere podatke iz naprave
82  clEnqueueReadBuffer(
83      command_queue, bufferResult, CL_TRUE, 0,
84      N * sizeof(int), (void*) result,
85      0, nullptr, nullptr);
86
87  // sprostimo vire
```

```
88 clReleaseCommandQueue(command_queue);
89 clReleaseMemObject(bufferResult);
90 clReleaseMemObject(bufferB);
91 clReleaseMemObject(bufferA);
92 clReleaseProgram(program);
93 clReleaseContext(context);
```

Izsek programa 2.3: Potrebni koraki za zagon OpenCL ščepca.

Kot lahko vidimo iz primera 2.3, je potrebno za izvedbo kratkega ščepca na ciljni napravi, pri gostitelju izvesti precejšnje število klicev OpenCL funkcij. Te funkcije nastavijo potrebne podatkovne strukture in preostalo okolje. Poleg tega lahko pri samih klicih zasledimo precejšnje število argumentov, kjer je poleg podane podatkovne strukture potrebno podajati še njeno velikost. Za parametre, ki jih ne potrebujemo pri našem klicu, moramo uporabiti „prazne“ argumente, kjer podamo vrednost `nullptr` ali `0`. Tak izgled funkcij lahko pripišemo dejstvu, da je bil OpenCL napisan v in za programski jezik C. Jezik C ne omogoča različnih funkcij z istim imenom, ki bi jih prevajalnik ločil po številu in tipu argumentov (angl. *function overloading*). S tem se ne samo oteži pisanje funkcij „na pamet“, ampak tudi poveča možnost vnosa hroščev in napak v program. To je prav tako eno izmed področij, na katerem SYCL in njegova uporaba jezika C++ programerju olajša delo.

2.2 SYCL

SYCL je standard [16] namenjen programiranju v heterogenih sistemih in temelji na prenosljivosti in učinkovitosti OpenCL, vendar omogoča programiranje v enotnem, visokonivojskem jeziku (C++11). To prinaša lažjo uporabo, saj programerju ni več potrebno skrbeti za veliko nizkonivojskih problemov, na katere je potrebno misliti pri programiranju v jeziku C. Prav tako se celotno SYCL aplikacijo napiše v enem programskem jeziku (C++), kjer je ščepec del preostale gostiteljske kode in ga ni potrebno pisati v posebnem

dialektu (OpenCL C). Ščepca tudi ni potrebno shranjevati v ločene datoteke ali pisati kot niza znakov, kot je to potrebno pri OpenCL.

Razvijalec lahko z uporabo obstoječih predlog (angl. *template*) pohitri in poenostavi razvoj aplikacij, z uporabo predlog algoritmov (angl. *algorithm template*) pa tudi poveča učinkovitost in hitrost izvajanja. Programerju se pri kreiranju osnovnih SYCL objektov, ki so potrebni za izvajanje, ni potrebno ukvarjati s podrobnostmi in nastavitvami gostiteljske in ciljne naprave, saj se le-te pogosto nastavijo samodejno. Kljub temu pa ima na voljo možnost, da vrednosti nastavi po svojih željah. S tem se zmanjša obseg napisane kode in posledično zmanjša verjetnost za vnos napak v program.

Kljub uporabi enega skupnega programskega jezika za pisanje tako gostiteljske kode kot ščepcev, pa standard dovoljuje uporabo enega ali pa več prevajalnikov za prevajanje. Prav tako za samo izvajanje ne zahteva OpenCL naprave, saj je v SYCLu gostiteljska naprava tudi ena izmed ciljnih naprav, ki se lahko uporabi v primeru, ko ni na voljo nobene druge OpenCL ciljne naprave. Tako se v primeru nedostopnosti naprave, ki podpira OpenCL, izvede prenos izvajanja ščepca nazaj na gostitelja (angl. *host fallback*). Ker gostiteljski napravi ni potrebno podpirati OpenCL, je lahko implementacija podpore *host fallback* realizirana z drugimi pohitritvenimi tehnologijami, kot je na primer OpenMP [11]. S prej opisanimi pristopi se olajša integracijo z obstoječimi orodji ter omogoča izbiro najprimernejšega prevajalnika glede na ciljno napravo [17].

SYCL uporablja obstoječe koncepte iz OpenCL. Tudi tu gostiteljska naprava pripravi vse potrebno za zagon ščepca na napravi, ciljna naprava pa ga izvede. Kot pri OpenCL pa tudi SYCL prinaša nekatere omejitve, večinoma zavoljo zmanjševanja odvisnosti med gostiteljsko in ciljno napravo. Standard v ščepcih ne dovoljuje uporabo kazalcev na funkcije, virtualnih funkcij (angl. *virtual functions*), informacije o tipih med izvajanjem (angl. *Runtime Type Information - RTTI*) ter knjižnic, ki uporabljajo te funkcije jezika.

SYCL cilja na uporabnike, ki si želijo hitrosti, učinkovitosti in prenosljivosti OpenCL, a hkrati tudi večjo fleksibilnost, ki jo pridobijo z uporabo

jezika C++.

2.3 Primerjava med OpenCL in SYCL

Kot je opisano v prejšnjih poglavjih, z uporabo SYCLa olajšamo pisanje paralelnega programa za heterogene sisteme.

Izsek 2.4 demonstrira kako se definira in zažene ščepec z enako funkcionalnostjo kot v izseku 2.2.

```
1  int a[N], b[N], result[N];
2
3  using namespace cl::sycl;
4
5  {
6    // vrsta za napravo tipa GPE
7    // avtomatsko izbere primerno platformo,
8    // napravo in kontekst
9    gpu_selector gpu;
10   queue q(gpu);
11
12   // v pomnilniku naprave
13   // rezervira in nastavi prostor
14   auto rN = range<1>(N);
15   auto bufferA = buffer<int>(a, rN);
16   auto bufferB = buffer<int>(b, rN);
17   auto bufferResult = buffer<int>(result, rN);
18
19   // cgh = command group handler
20   q.submit([&](handler& cgh)
21   {
22     // dostop do rezerviranega
23     // prostora na napravi
```

```
24      auto a = bufA.get_access<
25          access::mode::read,
26          access::target::global_buffer>(cgh);
27      auto b = bufB.get_access<
28          access::mode::read,
29          access::target::global_buffer>(cgh);
30      auto result = bufResult.get_access<
31          access::mode::write,
32          access::target::global_buffer>(cgh);
33
34      // ščepec
35      cgh.parallel_for<class example>(rN, [=](id<1> i)
36      {
37        result[i] = a[i] + b[i];
38      });
39    });
40  } // sinhronizacija vseh podatkov se izvede
41    // avtomatsko ko pride vrsta q izven
42    // območja vidnosti (scope)
```

Izsek programa 2.4: Potrebni koraki za zagon SYCL ščepca. Povzeto iz [22].

Takoj lahko opazimo, da je število korakov in klicev funkcij precej manjše, saj se z uporabo objektov precejšnje število nastavitev nastavi samodejno že pri sami konstrukciji objekta. Za razliko od OpenCL, v SYCLu ščepec podamo kot lambda funkcijo, ki je prav tako napisana v C++.

Jedro ščepca je praktično identično kot v OpenCL verziji algoritma opisanega v izseku 2.2. Obstaja pa razlika pri dostopu do vhodnih in ciljnih podatkov. Pri SYCLu namesto polj uporabljamo dostopne objekte (angl. *accessor object*), ki jih pridobimo iz buffer objektov, s katerimi ovijemo prvotne podatke (v tem primeru polja). Premikanje podatkov na in iz naprave se izvaja avtomatsko, pri tem pa SYCL sam postavi sinhronizacijske točke,

s katerimi skrbi za pravilnost prenesenih podatkov. Hkrati poizkuša s pametnim razporejanjem premikov in kopiranj čimbolj povečati učinkovitost in hitrost izvajanja.

Poglavje 3

Prenos sycl-gtx na Linux in koprocesor Intel Xeon Phi

3.1 sycl-gtx

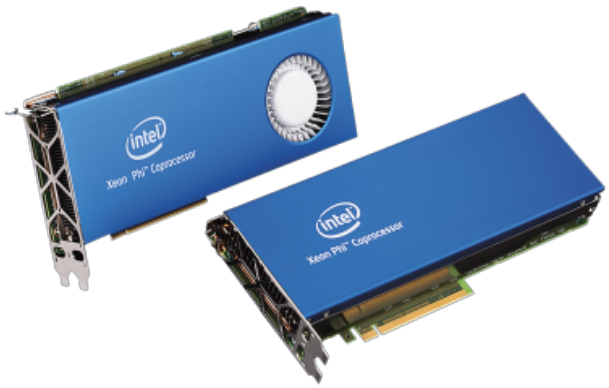
Za potrebe te naloge smo uporabili SYCL implementacijo z imenom sycl-gtx, ki jo je razvil Peter Žužek v okviru svoje magistrske naloge [22], in objavil za prosto uporabo [21] pod licenco MIT. Avtor se je za razvoj lastne implementacije odločil, ker do takrat ni obstajala nobena primerna odprta implementacija standarda. Odprtokodni triSYCL [15] je objavljen pod primerno permisivno licenco, vendar je njegov razvoj počasen, oziroma je zamrl. Codeplay Software, eden izmed glavnih pobudnikov standarda, sicer aktivno razvija svojo SYCL implementacijo, ki jo trži kot del programskega kompleta ComputeCpp [3], vendar pa je ta zaprtega tipa in pod komercialno licenco. Kljub temu pa je Peter Žužek s strani podjetja Codeplay uspel pridobiti testno licenco, s katero si je pomagal pri primerjavi s svojo implementacijo. Primerjavo z njegovimi testi opiše poglavje 4.

Sycl-gtx uporablja OpenCL 1.2 in navaden C++11 prevajalnik, saj se je razvoj samega ogrodja z uporabo obstoječih orodij poenostavil in pohitril. Implementacija specifikacije je bila zastavljena kot C++11 knjižnica, ki pri izvajanju za potrebe prevajanja ščepcev kliče lasten Just-In-Time (*JIT*) pre-

vajalnik. Kljub temu da sycl-gtx ne izpolnjuje vseh zahtev, ki jih nalaga standard, saj ga je razvijala le ena oseba v roku enega leta, pa je le dovolj razvit, da smo ga lahko uporabili kot osnovo za naš problem.

Prvotno je bil sycl-gtx razvit na operacijskem sistemu Microsoft Windows s pomočjo orodja Visual Studio, vendar pa je avtor kasneje dodal podporo za prevajanje kode na različnih platformah z orodjem CMake [2]. Kljub dodani podpori za prenos med sistemi, pa sam prenos ni bil nikoli izveden in testiran.

3.2 Ciljna naprava Intel Xeon Phi



Slika 3.1: Koprocesor Intel Xeon Phi [18].

Koprocesor Intel Xeon Phi [6] (slika 3.1) temelji na Intelovi arhitekturi MIC (*Many Integrated Core*), ki je bila razvita kot alternativa zmogljivim GPE in naj bi zasedala vmesno območje med CPE in GPE. Glavna razlika med Xeon Phi in GPE je v številu in arhitekturi jeder, saj ima Xeon Phi precej manjše število jeder, a so ta zmogljivejša kot tista v grafičnih enotah. Jedra podpirajo 64-bitne ukaze široko razširjene arhitekture Intel x86. Posamezno jedro v koprocesorju Xeon Phi je tako precej bolj podobno tistim, ki jih najdemo v CPE, vendar pa le-ta vsebujejo tudi 512 bitno vektorsko enoto. Ta enota omogoča vse osnovne matematične operacije, poleg tega pa podpira vektorske operacije - to je sočasno seštevanje in množenje več števil

naenkrat. Intelov koprocesor vsebuje tudi več predpomnilnika kot GPE, ter s pomočjo tehnologije *HyperThreading* izvaja do štiri niti na enem jedru.

Kljub temu, da Xeon Phi podpira ukaze x86 in na njem teče operacijski sistem Linux, pa na gostitelju vseeno potrebuje vsaj en procesor, ki skrbi za dejansko izvajanje vseh ostalih potrebnih storitev (glavni operacijski sistem, prevajalnik...).

Xeon Phi omogoča dva načina izvajanja prevedene kode:

1. *Način razbremenitve* (angl. *„Offload“*). Aplikacija se prevede in zažene na gostiteljskem sistemu, ki tudi upravlja izvajanje, izbrani deli kode pa se izvajajo na koprocesorju - na primer ščepci v OpenCL ter sekcije `#pragma offload` v OpenMP.

2. *„Koprocesorski“ način.* Aplikacijo na gostiteljskem sistemu prevedemo z Intelovim C/C++ prevajalnikom `icc`, kateremu poleg ostalih potrebnih opcij dodamo še opcijo `-mmic`. S tem se program prečno prevede (angl. *cross compile*) za Xeon Phi. Prevedeno izvršljivo datoteko skopiramo na koprocesor, kjer jo zaženemo.

3.3 Ciljni sistem

Glavni računalnik na katerega smo prenašali obstoječo implementacijo syclgtx je strežnik z imenom `gpufarm`, ki se nahaja na Fakulteti za računalništvo in informatiko. Na njem teče GNU/Linux operacijski sistem CentOS 6.8. Strojna oprema strežnika je sestavljena iz dveh strežniških procesorjev Intel Xeon E5-2620 [6], kjer ima vsak posamezen procesor po šest jeder z osnovno frekvenco 2,0 GHz, ki pa se lahko pri zahtevnejših izračunih poviša do 2,5 GHz. Vsak procesor omogoča sočasno izvajanje 12 niti, saj se lahko na posameznem jedru zaradi tehnologije *HyperThreading* izvajata 2 niti. Na obeh procesorjih je torej na voljo skupno 24 niti. Na razpolago ima tudi 62 GB glavnega pomnilnika. Poleg CPE, OpenCL podpirajo tudi prej omenjen koprocesor Intel Xeon Phi ter dve grafični kartici namenjeni računanju, Nvidia

Tesla K20 [9] (slika 3.2), s 2496 jedri, ki tečejo pri frekvenci 706 MHz ter 5 GB pomnilnika tipa GDDR5.

Slika 3.2: GPE Nvidia Tesla K20.

3.4 Prenos kode

Da smo lahko sycl-gtx pognali na strežniku `gpufarm` je bilo prej potrebno odpraviti nekaj pomanjkljivosti. Na strežniku je bilo potrebno posodobiti sistem za avtomatizirano generiranje datotek, ki skrbijo za prevajanje *CMake* [2], saj obstoječa verzija ni vsebovala potrebnih skript za avtomatsko dodajanje potrebnih knjižnic pri prevajanju. Prav tako je bilo potrebno poiskati C++ prevajalnik, ki je podpiral standard C++11. Na strežniku je bil poleg privzetega prevajalnika `g++` verzije 4.4 naložen tudi `g++` verzije 4.8, ki je podpiral zahtevan standard.

Poleg izvajanja in merjenja testov, opisanih v poglavju 4, smo poizkusili tudi nekatere druge načine prevajanja in izvajanja.

3.4.1 Drugi načini izvajanja

Zadali smo si cilj poizkusiti ali je mogoče sycl-gtx prevesti in pognati na koprocesorju v koprocesorskem načinu, omenjenem v poglavju 3.2.

Pri prevajanju z opcijo `-mmic` smo najprej naleteli na težavo, pri kateri nam je Intelov C/C++ prevajalnik (`icc`) javljal napake pri prevajanju delov

kode, ki jih v jezik prinaša standard C++11. Najprej smo pomislili, da je na `gpufarm` naložena prestara verzija `icc` prevajalnika, ki morda podpira le nekatere zahteve standarda C++11, vendar pa so napake pri prevajanju ostale tudi po posodobitvi licence in verzije prevajalnika na najnovejšo verzijo 17. Ugotovili smo, da se `icc` naslanja na GNUjeva prevajalnika `gcc` in `g++` ter pripadajoče zaglavne datoteke in knjižnice. Tu je nastal problem, saj je `icc` zaradi različnih razpoložljivih verzij `g++` prevajalnikov uporabil prestaro (privzeto) verzijo. Z opcijama `-gcc-name=gcc48` in `-gxx-name=g++48` (kjer sta `gcc48` in `g++48` verziji 4.8 C/C++ prevajalnika) pa so napake pri prevajanju zaradi standarda C++11 izginile. Kljub razrešitvi prej omenjenih napak, pa smo ugotovili, da Xeon Phi v koprocesorskem načinu ne podpira OpenCL kode. To smo ugotovili, ker prevajalnik `icc` pri prevajanju ni našel potrebnih zaglavnih datotek, ki jih potrebuje za prečno prevajanje za Xeon Phi, kar nam je prevajalnik javil z sledečo napako:

```
catastrophic error: cannot open source file "CL/cl.h"
  #include <CL/cl.h>
                    ^
```

Kljub podajanju različnih poti do različnih verzij OpenCL knjižnic na strežniku nam napake ni uspelo odpraviti, iz česar sklepamo, da knjižnice (in podpora za OpenCL na Xeon Phi) ne obstajajo. Naše sume glede neobstoječe podpore za OpenCL v koprocesorskem načinu je potrdilo tudi pomanjkanje podobnih zapisov in primerov na spletu.

Prav tako nas je zanimalo, ali je mogoče problem na preprost način še dodatno paralelizirati s hkratnim izvajanjem na dveh GPE, saj strežnik `gpufarm` vsebuje dve GPE za računanje Nvidia Tesla K20. Ugotovili smo, da sycl-gtx nima te podpore, prav tako pa je nima tudi sam OpenCL - problem bi bilo potrebno ročno razdeliti na dva dela.

Poglavje 4

Testiranja in meritve

4.1 Testna okolja

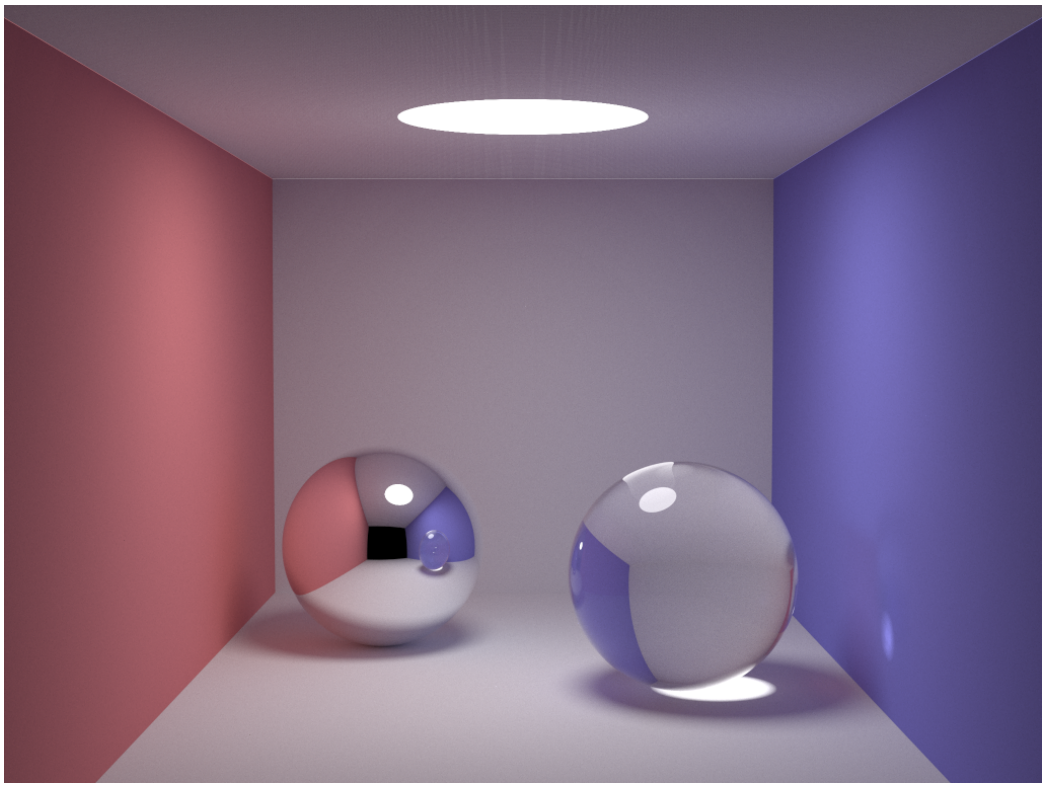
Testni primer `smallpt` opisan v poglavju 4.2 smo pognali na več računalnikih in na različnih napravah, z namenom primerjati hitrosti izvajanja enako zahtevnega problema na različni strojni opremi. Testirali smo na sledečih računalnikih s sledečimi napravami:

1. Strežnik `gpufarm` opisan v poglavju 3.3. Uporabili smo sledeče načine izvajanja:
   - CPE Intel Xeon E5-2620 brez paralelizacije,
   - CPE Intel Xeon, paralelizacija s pomočjo OpenMP [11],
   - CPE Intel Xeon, paralelizacija z OpenCL,
   - GPE Nvidia Tesla K20 z OpenCL,
   - koprocesor Intel Xeon Phi z OpenCL.
2. Prenosnik višjega cenovnega razreda z operacijskim sistemom Linux:
   - CPE Intel Core i7-5600U [5] z osnovno frekvenco 2,6 GHz, najvišjo frekvenco 3,2 GHz in 2 jedroma (ki zaradi tehnologije HyperThreading omogočata izvajanje skupno 4 niti), brez paralelizacije,

- CPE Intel Core i7-5600U, paralelizacija z OpenMP,
- Integrirana GPE Intel HD 5500 z OpenCL.

3. Stacionarni računalnik srednjega cenovnega razreda z operacijskim sistemom Linux:
   - CPE AMD Phenom II X4 955 [4] s frekvenco 3,2 GHz in 4 jedri, brez paralelizacije,
   - CPE AMD Phenom II X4 955, paralelizacija z OpenMP.

4.2 Testni primer smallpt



Slika 4.1: Rezultat testnega programa smallpt.

Poleg implementacije standarda SYCL, sycl-gtx sestavlja tudi nekaj testnih programov. Najobsežnejši od njih je odprtokodni sledilec žarkov (angl.

ray tracer) `smallpt` (rezultat izvajanja na sliki 4.1), ki je bil prvotno objavljen na Codeplayovem blogu [19]. Avtor sycl-gtx, Peter Žužek je prvotno implementacijo prenesel v sycl-gtx (izsek 4.1, brez vsebine ščepca), ter dodal kodo ki olajšuje testiranje na platformah, napravah in različnih stopnjah podrobnosti.

Sledenje žarkov (angl. *ray tracing*) je način generiranja slik, pri kateri algoritem simulira potek žarka skozi kompozicijo, pri tem pa upošteva kaj se zgodi z žarkom, ko zadane nek objekt. Tak način generiranja slike prinaša visoko realističnost videza slike, vendar pa je računsko zelo zahteven. Zato je primeren le za uporabo pri problemih, kjer je dovolj časa za vnaprejšnjo izdelavo slike. Večje število vzorcev na slikovno točko (angl. *pixel*) prinaša boljše rezultate in večjo realističnost slike.

```
1  void compute_sycl_gtx(
2    void* device,
3    int width, int height, int samplesPerPixel,
4    Ray cameraRay, Vec cxIn, Vec cyIn,
5    Vec initialRadiance, Vec* colors)
6  {
7    {
8      // vhodne podatke vstavimo v SYCL
9      // podatkovne strukture
10     buffer<Vec, 1> color_buffer(
11         colors, range<1>(width * height));
12     buffer<Sphere, 1> spheres_buffer(
13         &spheres_global[0], range<1>(9));
14
15     auto commandGroup = [&](handler& cgh) {
16       kernel_r smallpt = {
17         // dostop do podatkov na napravi
18         color_buffer.get_access<
19             access::mode::write>(cgh),
```

```
20          spheres_buffer.get_access<
21              access::mode::read,
22              access::target::constant_buffer>(cgh),
23          width, height, samplesPerPixel,
24          cameraRay, cxIn, cyIn, initialRadiance
25        };
26        nd_range<2> ndr(
27            range<2>(width, height), range<2>(8, 8));
28        cgh.parallel_for(ndr, smallpt);
29      };
30      // postavimo v vrsto za zagon
31      q.submit(commandGroup);
32    }
33  }
```

Izsek programa 4.1: Zagon smallpt SYCL ščepca (brez vsebine ščepca) [22].

4.3 Meritve

Izvajalne čase smo merili na napravah opisanih v poglavju 4.1. Isti testni problem (`smallpt`) smo na napravah pognali večkrat, pri vsaki iteraciji pa smo povečevali število vzorcev na slikovno točko. S tem se je povečevala kvaliteta generirane slike ter podaljšal izvajalni čas.

Poleg merjenja izvajalnih časov na naših ciljnih napravah, pa smo opravili tudi primerjave s pripadajočimi meritvami Petra Žužka v njegovi magistrski nalogi [22]. Pri svojih meritvah je uporabljal CPE Intel Core i5-4570 s 4 jedri in frekvenco 3,2 GHz ter vgrajeno GPE Intel HD Graphics 4600.

Tabela 4.1 vsebuje rezultate izvajanja testa `smallpt` na strežniku `gpufarm`, tabela 4.2 pa rezultate izvajanja na prenosniku in stacionarnem računalniku. Sliki 4.2 in 4.3 grafično prikažeta razlike v izvajalnih časih pri uporabi sycl-gtx. Za primerjavo z našimi meritvami je dodana še tabela 4.4, ki vsebuje

meritve, ki jih je izvedel Peter Žužek v svoji nalogi, in sicer izvajanje brez uporabe SYCLa, paralelizacija z uporabo sycl-gtx in paralelizacija z uporabo ComputeCpp [3].

Poleg izvajalnih časov, so v tabeli 4.3 in na slikama 4.4 in 4.5 zapisane še pohitritve pri izvajanju SYCLa v primerjavi z izvajanjem sekvenčnega algoritma. Vsebujejo podatke iz naprav, ki podpirajo SYCL oziroma OpenCL.

Sliki 4.6 in 4.7 prikažeta razlike v časih izvajanja pri paralelizaciji z OpenMP na strežniku `gpufarm`, na prenosniku in stacionarnem računalniku, dodane pa so tudi vrednosti, ki jih je dobil Peter Žužek v svoji nalogi s tem načinom paralelizacije. Sliki 4.8 in 4.9 vsebujeta pohitritve v primerjavi s sekvenčnim algoritmom na prej naštetih napravah.

Dodani so tudi grafi, ki primerjajo razlike v hitrostih izvajanja med našimi meritvami in meritvami v nalogi Petra Žužka. Sliki 4.10 in 4.11 primerjata meritve na CPE brez paralelizacije, sliki 4.12 in 4.13 primerjata meritve na CPE pri paralelizaciji s SYCLom. Sliki 4.14 in 4.15 prikažeta meritve na GPE pri paralelizaciji s SYCLom, kjer smo pri meritvi paralelnih testov uporabili sycl-gtx, Peter Žužek pa sycl-gtx in ComputeCpp.

Vse meritve so v sekundah, pohitritve pa v odstotkih. Podatki v tabelah so razvrščeni po številu vzorcev, ki so bili uporabljeni pri generiranju, ter po različnih računalnikih ter njihovih napravah, oziroma tehnologijah, ki so bile uporabljene.

Pohitritve so izračunane po enačbi

$$S = \frac{t_s}{t_p}$$

kjer je S pohitritev, t_s čas izvajanja sekvenčnega algoritma, t_p pa čas izvajanja paralelnega algoritma.

Število vzorcev	Intel Xeon	Intel Xeon + OpenMP	Intel Xeon + SYCL	Intel Xeon Phi + SYCL	Nvidia Tesla K20 + SYCL
4	35,046	15,642	2,300	4,677	1,520
8	69,725	32,124	3,040	5,777	2,340
16	138,958	63,931	5,490	10,147	3,973
32	277,693	124,923	10,806	16,258	6,583
64	555,162	249,633	20,693	30,888	11,883
128	1114,190	489,543	39,871	57,876	24,515
256	2228,160	1007,730	79,400	114,533	51,828
512	4455,560	2009,450	156,077	224,222	85,688
1024	8911,580	4023,852	313,661	446,845	116,305
2084	17822,100	8055,370	627,070	892,682	175,230
4096			1234,310	1776,760	295,350
8192			2492,850	3560,040	532,021

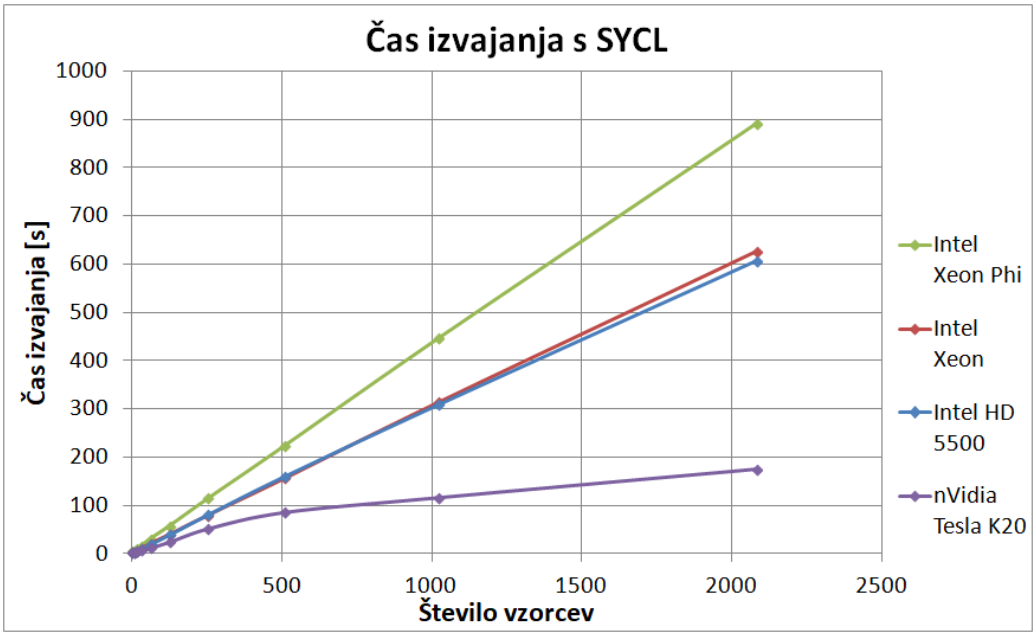
Tabela 4.1: Meritve testa smallpt s sycl-gtx, izvedene na strežniku gpufarm. Vrednosti so v sekundah.

Število vzorcev	Prenosnik			Stacionarni računalnik	
	Intel i7-5600U	Intel i7-5600U + OpenMP	Intel HD 5500 + SYCL	AMD Phenom II	AMD Phenom II + OpenMP
4	21,896	9,960	1,622	40,087	13,932
8	43,610	19,169	2,670	80,058	27,809
16	87,807	38,271	5,219	159,968	55,506
32	175,023	77,540	10,008	319,808	110,961
64	347,767	156,588	19,640	639,342	221,874
128	694,452	314,341	39,710	1278,740	443,698
256	1388,590	631,048	81,228	2556,970	902,173
512	2777,650	1262,220	160,855	5113,400	1775,270
1024	5564,560	2517,980	309,525	10598,732	3552,020
2084	11114,400	5009,830	607,627		7103,160
4096			1098,930		
8192			1253,450		

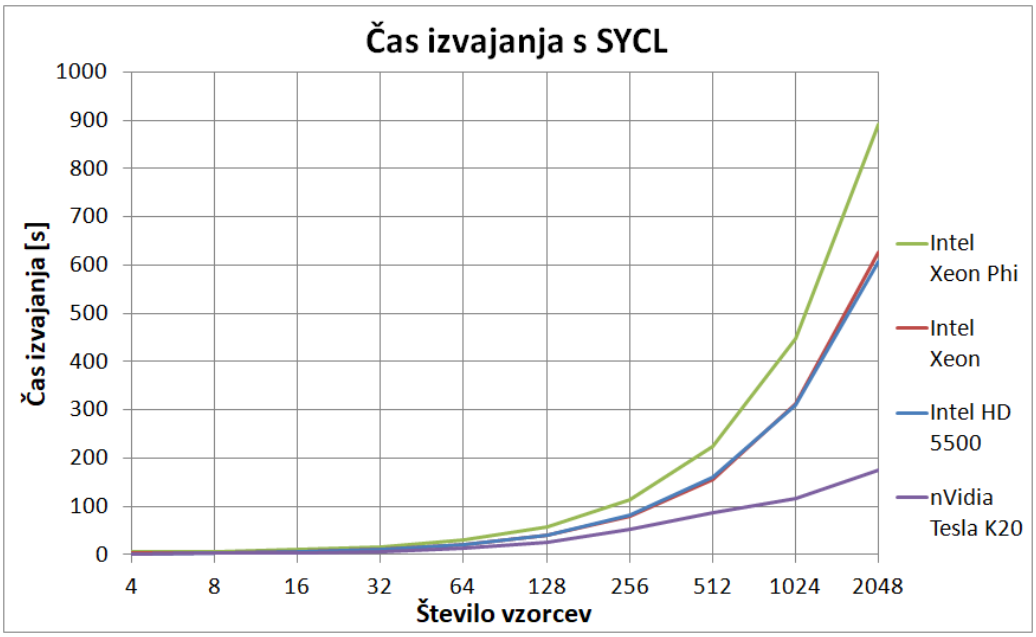
Tabela 4.2: Meritve testa smallpt s sycl-gtx, izvedene na prenosniku in stacionarnem računalniku. Vrednosti so v sekundah.

Število vzorcev	Strežnik gpufarm			Prenosnik
	Intel Xeon + SYCL	Intel Xeon Phi + SYCL	Nvidia Tesla K20 + SYCL	Intel HD 5500 + SYCL
4	15,24	7,49	23,06	13,50
8	22,93	12,07	29,79	16,33
16	25,31	13,69	34,97	16,82
32	25,70	17,08	42,18	17,48
64	26,83	17,97	46,72	17,70
128	27,95	19,25	45,45	17,48
256	28,06	19,45	42,99	17,09
512	28,55	19,87	52,00	17,26
1024	28,41	19,94	76,62	17,97
2084	28,42	19,96	101,71	18,29

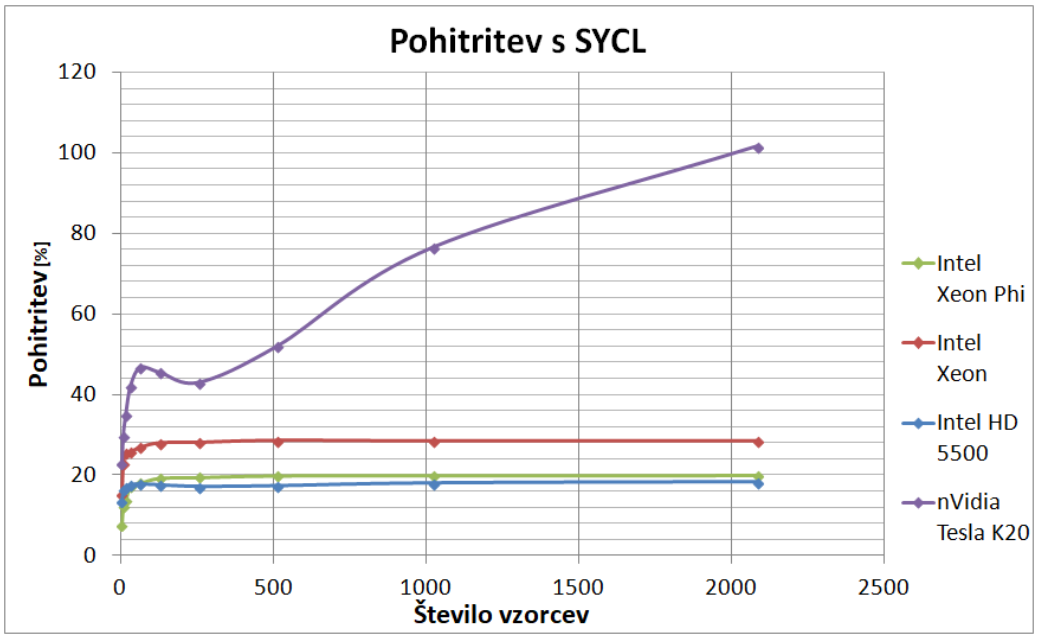
Tabela 4.3: Pohitritve testa smallpt s sycl-gtx na testnih napravah. Pohitritve so v odstotkih v primerjavi s prvotnem sekvenčnem programom zagnanim na CPE.



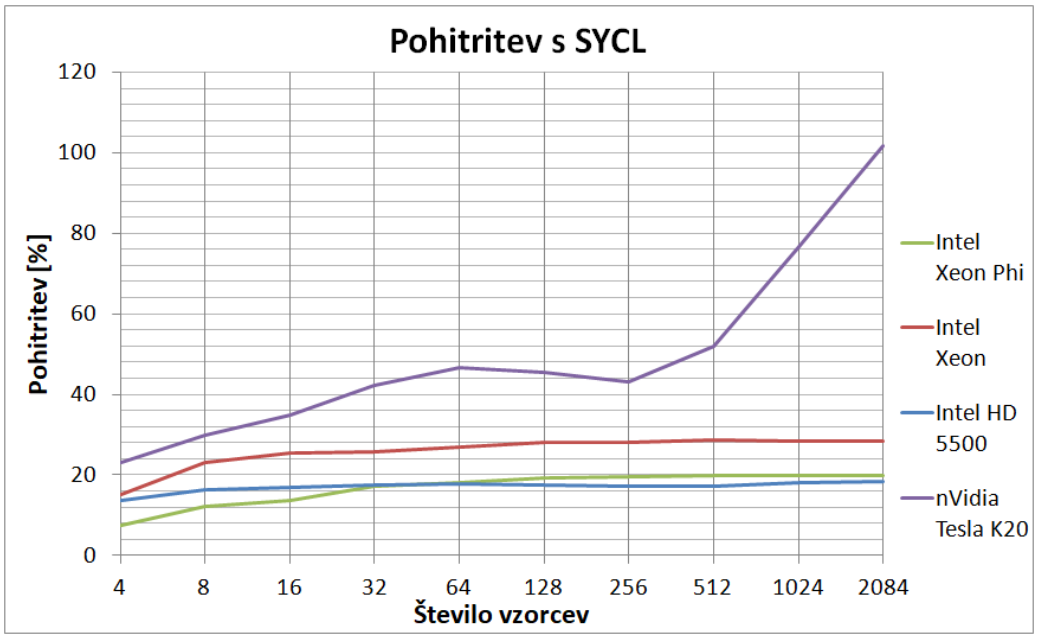
Slika 4.2: Čas izvajanja testa smallpt s sycl-gtx. Vrednosti x osi so porazdeljene linearno.



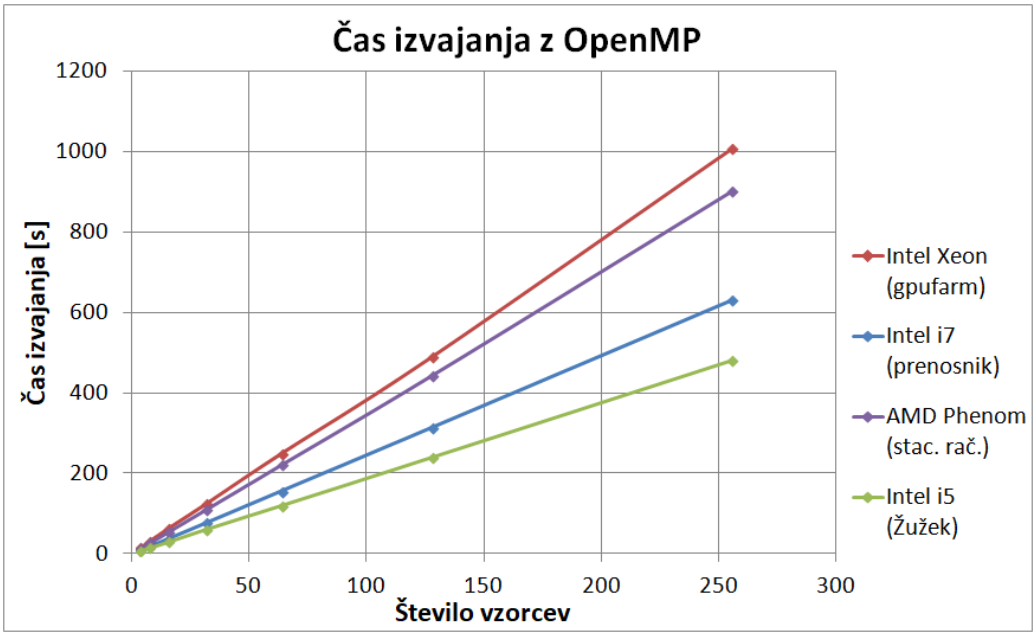
Slika 4.3: Čas izvajanja testa smallpt s sycl-gtx. Vrednosti x osi so porazdeljene logaritemsko.



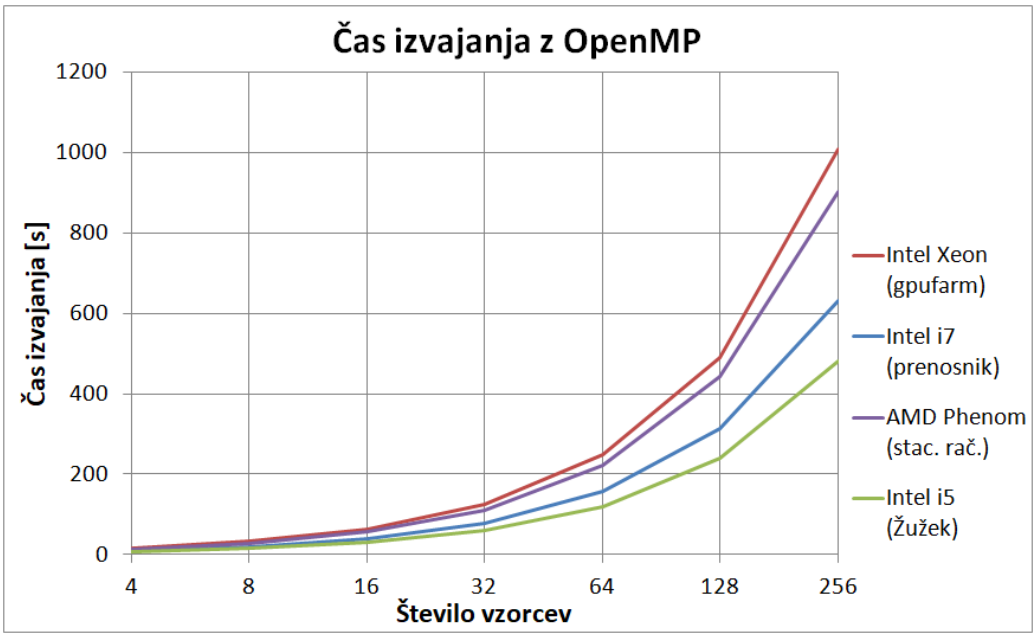
Slika 4.4: Pohitritve testa smallpt s sycl-gtx. Vrednosti x osi so porazdeljene linearno.



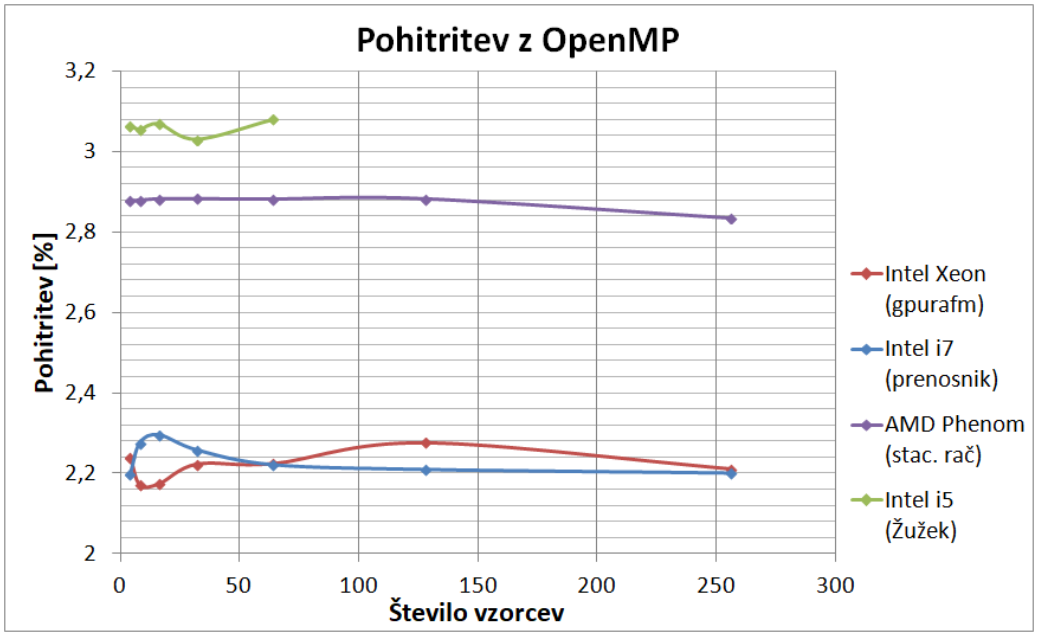
Slika 4.5: Pohitritve testa smallpt s sycl-gtx. Vrednosti x osi so porazdeljene logaritemsko.



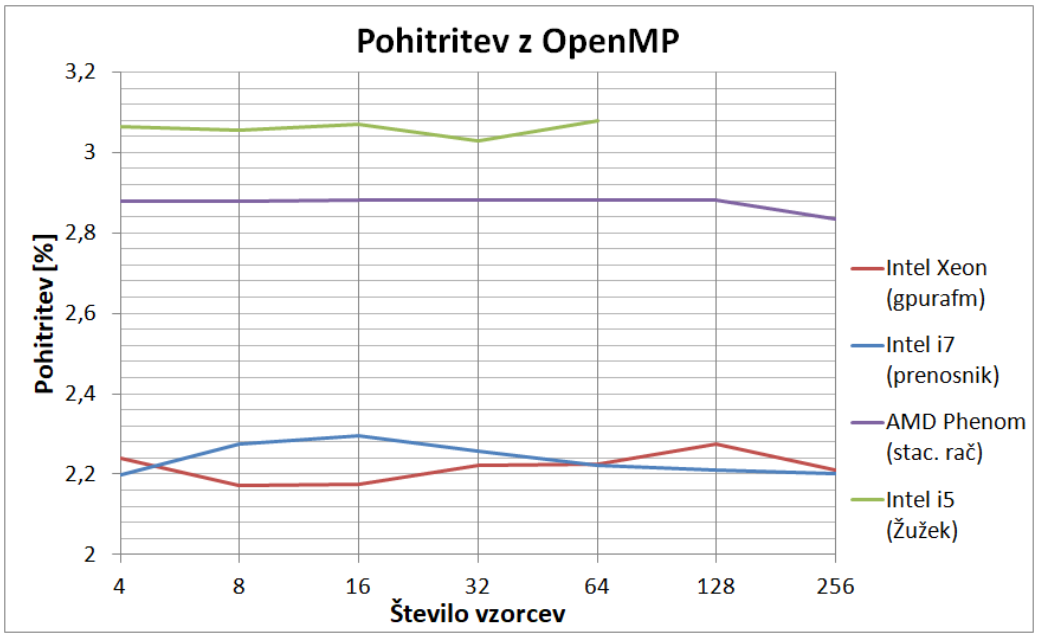
Slika 4.6: Čas izvajanja testa smallpt s sycl-gtx pri paralelizaciji z OpenMP. Vrednosti x osi so porazdeljene linearno.



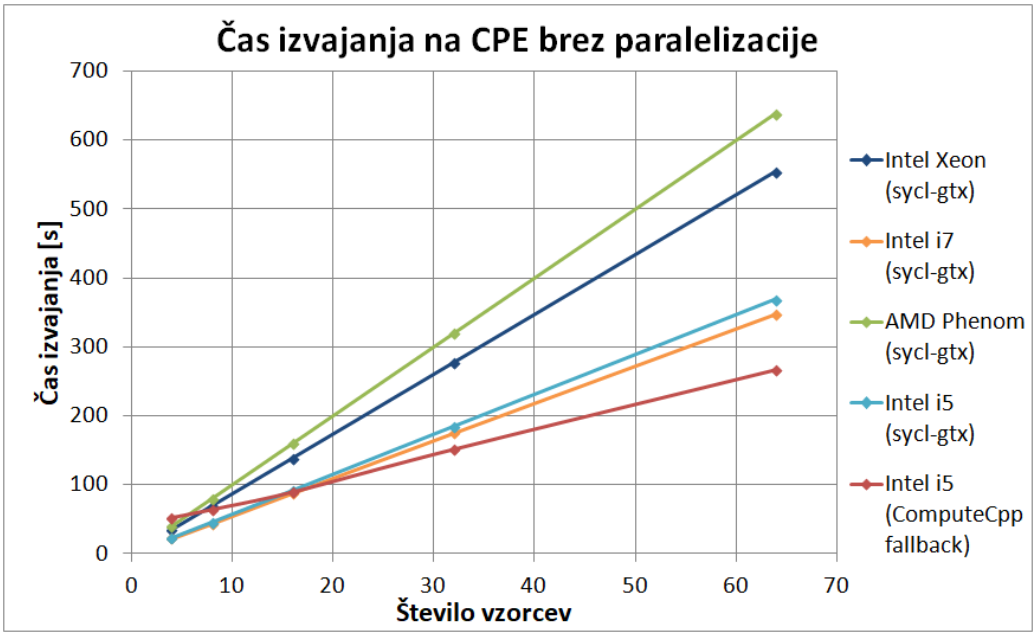
Slika 4.7: Čas izvajanja testa smallpt s sycl-gtx pri paralelizaciji z OpenMP. Vrednosti x osi so porazdeljene logaritemsko.



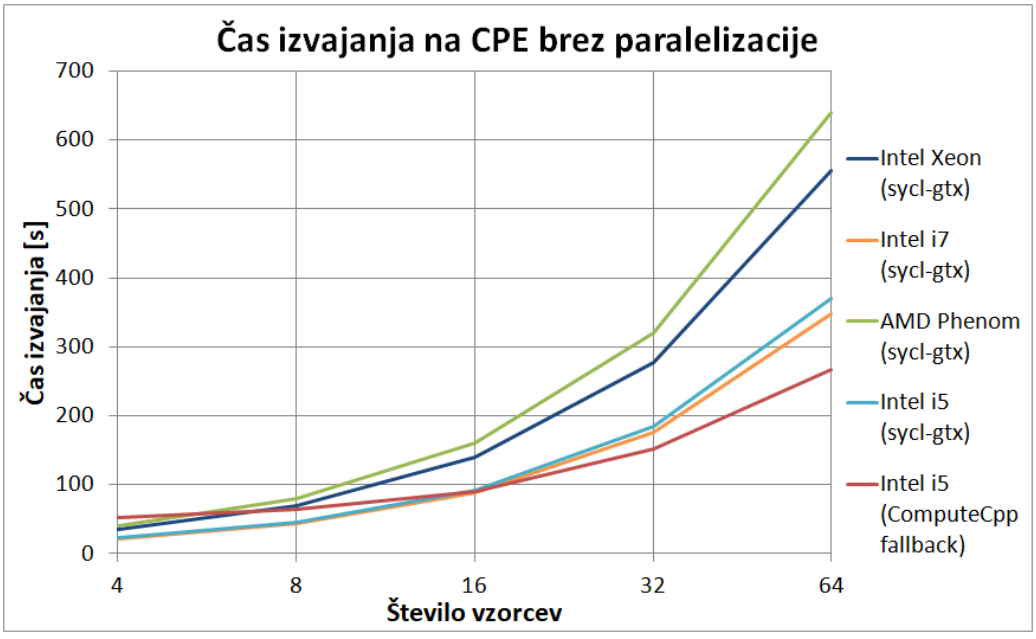
Slika 4.8: Pohitritev testa smallpt s sycl-gtx pri paralelizaciji z OpenMP. Vrednosti x osi so porazdeljene linearno.



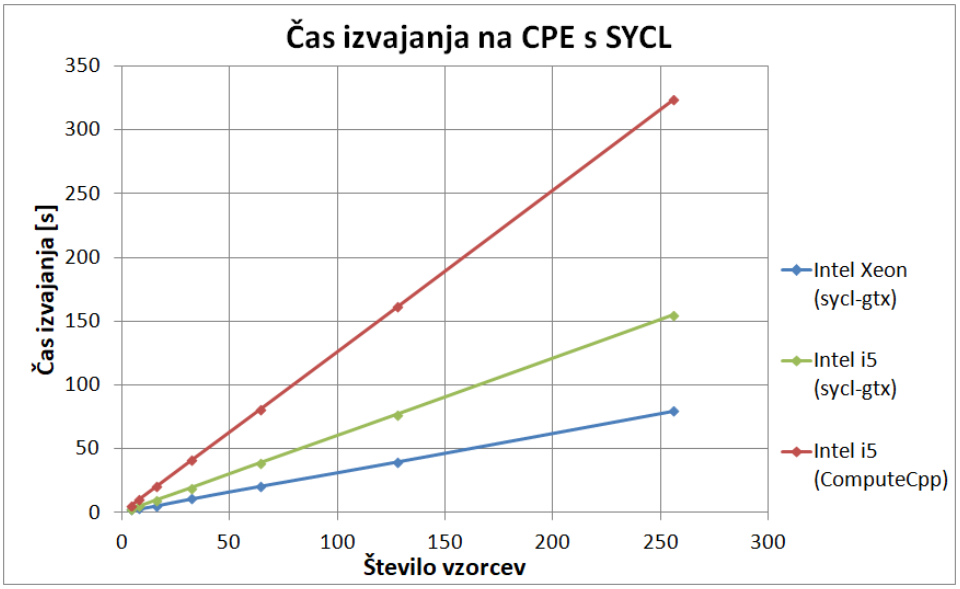
Slika 4.9: Pohitritev testa smallpt s sycl-gtx pri paralelizaciji z OpenMP. Vrednosti x osi so porazdeljene logaritemsko.



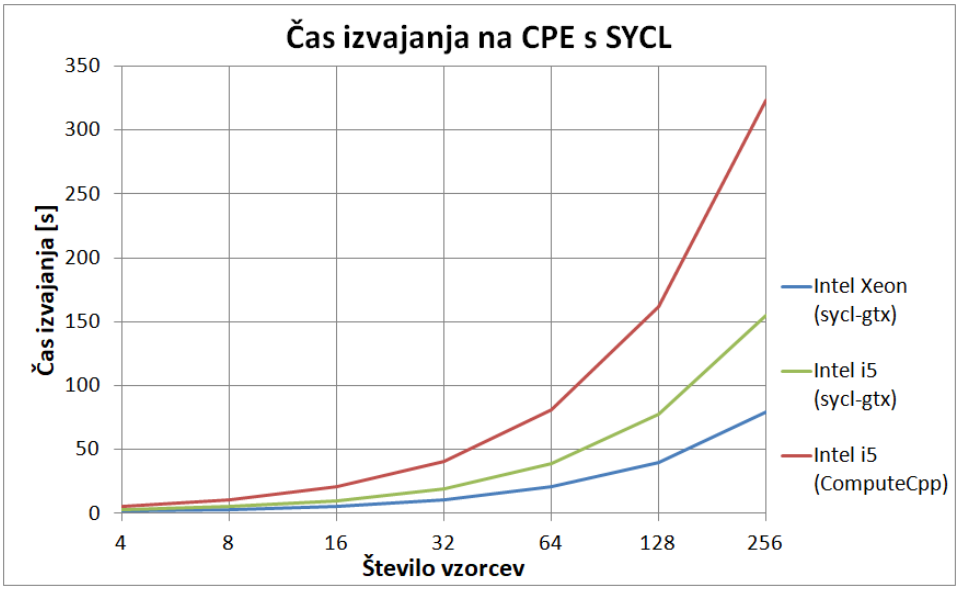
Slika 4.10: Primerjava naših meritev časa izvajanja sekvenčnih algoritmov z meritvami Petra Žužka [22]. Vrednosti x osi so porazdeljene linearno.



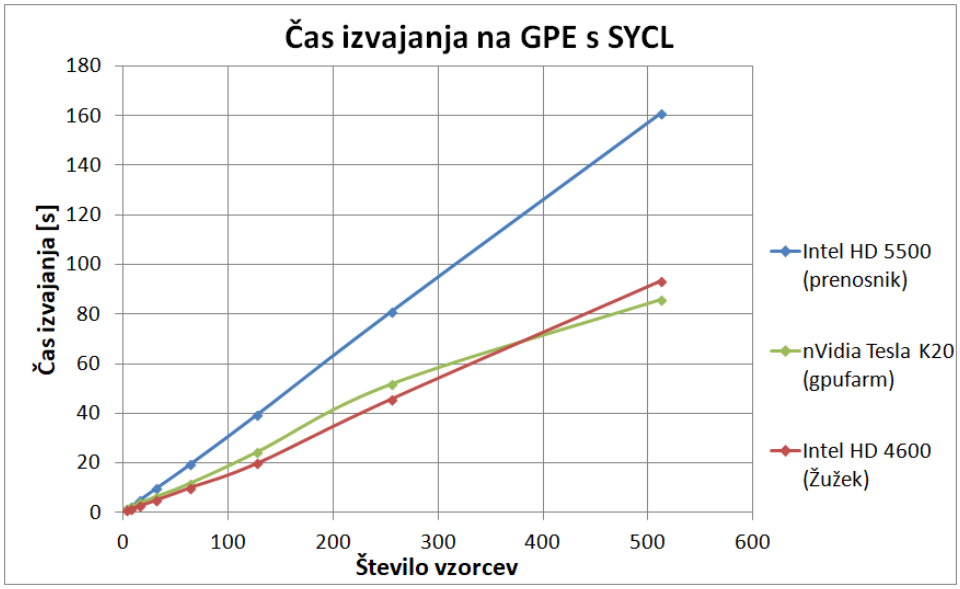
Slika 4.11: Primerjava naših meritev časa izvajanja sekvenčnih algoritmov z meritvami Petra Žužka [22]. Vrednosti x osi so porazdeljene logaritemsko.



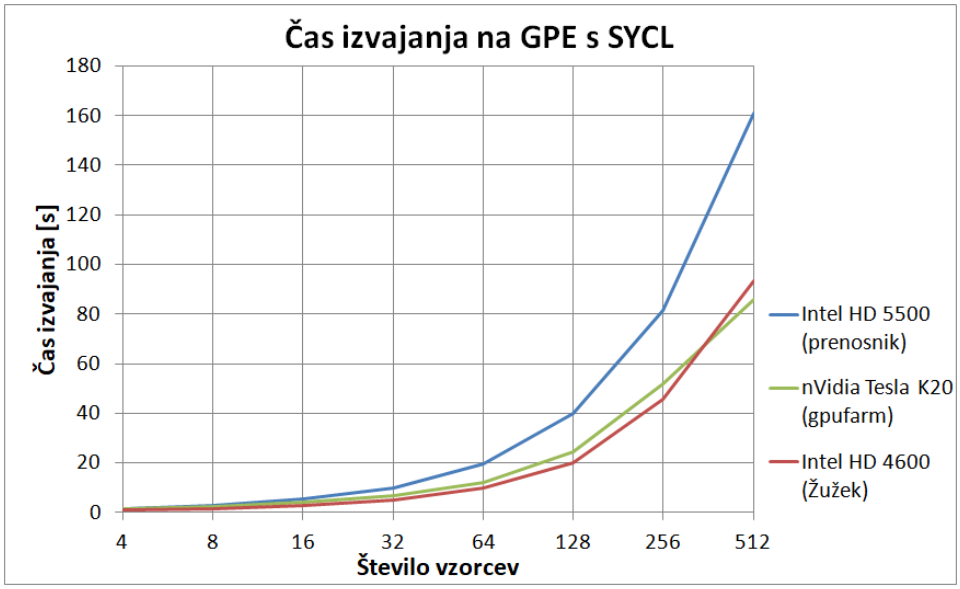
Slika 4.12: Primerjava naših meritev časa izvajanja na CPE pri paralelizaciji z SYCL z meritvami Petra Žužka [22]. Vrednosti x osi so porazdeljene linearno.



Slika 4.13: Primerjava naših meritev časa izvajanja na CPE pri paralelizaciji z SYCL z meritvami Petra Žužka [22] Vrednosti x osi so porazdeljene logaritemsko.



Slika 4.14: Primerjava naših meritev časa izvajanja na GPE pri paralelizaciji z SYCL z meritvami Petra Žužka [22]. Vrednosti x osi so porazdeljene linearno.



Slika 4.15: Primerjava naših meritev časa izvajanja na GPE pri paralelizaciji z SYCL z meritvami Petra Žužka [22]. Vrednosti x osi so porazdeljene logaritemsko.

Št. vz.	Brez SYCL		sycl-gtx		ComputeCpp	
	Intel i5-4570	Intel i5-4570 + OpenMP	Intel i5-4570 + SYCL	Intel HD 4600 + SYCL	Intel i5-4570 + SYCL	Intel i5-4570 + host fallback
4	23,02	7,51	2,95	1,04	5,48	51,86
8	45,93	15,03	5,27	1,61	10,52	63,81
16	91,82	29,91	10,1	2,75	20,67	89,29
32	184,76	60,99	19,58	5,11	40,77	151,52
64	369,24	119,87	39,03	10,03	80,78	267,52
128		239,21	77,24	19,9	161,4	501,24
256		479,7	154,64	45,79	323,37	
512			309,89	93,17		

Tabela 4.4: Primerjava izvajalnih časov izmerjene s strani Petra Žužka [22]. Časi so v sekundah.

4.4 Ugotovitve

Ker je problem `smallpt` pisan na kožo GPE, ne preseneča dejstvo, da so se le-te tudi najbolje odrezale. V primerjavi z ostalimi napravami na istem računalniku, sta bili GPE za nekaj redov boljši. Pričakovano se je najbolje odrezala GPE Nvidia Tesla K20, saj je le-ta tudi namenjena računanju. Na grafoma 4.4 in 4.5 pa lahko vidimo, da pri 128, 256 in 512 vzorcih naraščanje hitrosti izvajanja upade. Podobno prekinitev naraščanja smo opazili tudi po več ponovitvah testa. Domnevamo, da je vzrok v tem, da je pred tem problem prelahek in pride do anomalij pri meritvah, ki pa se izgubijo pri daljšem izvajanju.

Pri uporabi SYCL se je prav tako dobro izkazala CPE Intel Xeon. Poleg GPE je najhitreje izvajala izračune, saj ima za CPE razmeroma veliko število niti, velik predpomnilnik in velik glavni pomnilnik. Pri paralelizaciji s SYCL je sistem avtomatsko uporabil oba procesorja ki sta na voljo in izvajal izračune s 24 nitmi (6 jeder na procesor, 2 niti na jedro zaradi tehnologije

HyperThreading).

Pri uporabi OpenMP se je med našimi napravami presenetljivo najslabše odrezal Intel Xeon, najbolje pa Intel i7 (sliki 4.6 in 4.7). Poleg tega, da procesorska jedra na Intel Xeon pri polni obremenitvi dosegajo nižjo frekvenco kot na Intel i7 in AMD Phenom II, jih OpenMP tudi izkoristi bolj konservativno. Za izračun je na Intel Xeon uporabil le 18 niti (od možnih 24). S pomočjo orodja za preverjanje uporabe CPE, `htop`, smo opazili da samo nekatere niti dosežejo 100% izkoristek posameznega jedra, večji del niti pa dosega slabši izkoristek.

Boljše rezultate smo morda pričakovali s strani koprocesorja Intel Xeon Phi, ki bi lahko na račun arhitekture MIC, ki je nekakšen kompromis med navadnimi CPE in GPE, ter razmeroma velikim številom niti (236) prikazal boljše rezultate. Koprocesorju pa zagotovo ni v prid nižja frekvenca jeder v primerjavi s CPE (1 GHz proti 2 GHz). Verjamemo, da bi se dalo rezultate koprocesorja izboljšati, če bi kodo popravili na tak način, da bi bolje izkoristili posebnosti arhitekture MIC. To pa z vidika prenosljivosti sycl-gtx seveda ni smiselno, saj bi s tem zmanjšali prenosljivost in neodvisnost kode od naprav na katerih teče.

Za solidno se je izkazala GPE Intel HD 5500, ki je glede na to, da je del prenosniške CPE, zadan problem izračunala v nadvse sprejemljivem času. Dosegla je praktično enake izvajalne čase kot strežniška CPE Intel Xeon, ter podobne pohitritve kot koprocesor Xeon Phi.

Presenetila nas je primerjava z rezultati, ki jih je pri testiranju sycl-gtx dobil njegov avtor Peter Žužek, prikazani v tabeli 4.4. Ti so bili namreč ponekod občutno boljši, kljub temu, da smo mi uporabili opremo, ki naj bi bila vsaj v teoriji zmogljivejša. Dober primer je izvajanje na GPE (slika 4.15), kjer je njegovim rezultatom lahko konkurirala le GPE namenjena računanju (Nvidia Tesla), prenosniška GPE pa je bila za več kot en razred počasnejša. Primerjava GPE preko suhoparnih specifikacij sicer ne poda popolnoma realnega in objektivnega stanja, vendar nam lahko pomaga pri okvirni oceni razlik med napravami. V tabeli 4.5 lahko vidimo, da je Nvidia Tesla občutno

zmogljivejša, saj ima boljše vrednosti v praktično vseh kategorijah. Ogromna razlika pa je tudi v najbolj „praktični“ kategoriji - število izračunov v plavajoči vejici na sekundo (angl. *floating point operations per second - FLOPS*). Prav tako, so bili njegovi rezultati precej boljši pri izvajanju na CPE z uporabo OpenMP (grafa 4.7 in 4.9), kjer je prvotni avtor dobil znatno boljše rezultate v primerjavi z vsemi CPE, ki smo jih uporabili mi.

	Nvidia Tesla K20	Intel HD 4600	Intel HD 5500
Število jeder	2496	4	4
Najvišja frekv. jedra	706 MHz	1250 MHz	1000 MHz
Frekvenca spomina	2,6 GHz	800 MHz	/
Spominska pasovna širina	208 GB/s	25,6 GB/s	/
Količina spomina	5 GB	1,72 GB	2,1 GB
Računske enote (angl. *compute units*)	13	20	24
FLOPS	3,52 Tflops	288-432 Gflops	/

Tabela 4.5: Primerjava uporabljenih GPE [22], [14], [20].

Nad odličnimi dobljenimi rezultati pridobljenimi z njegovo implementacijo sycl-gtx, je bil presenečen tudi Peter Žužek, še posebej ko jih je primerjal s komercionalno implementacijo iz kompleta ComputeCpp (tabela 4.4). Prišel je do zaključka, da je po vsej verjetnosti nekje v implementaciji ali pri testiranju prišlo do napake pri programiranju, saj je ComputeCpp precej bolj razvita in preverjena implementacija standarda in je zaradi tega po vsej verjetnosti bolj zanesljiva.

Svoje ugotovitve je komentiral: „*We observed something very strange in the ComputeCpp OpenCL results: the values are almost exactly twice the sycl-gtx values. In fact, if we shift the ComputeCpp values one column to the right, we can observe a very small difference in results, shown in Figure 5.4 [v našem primeru sta to 4. in 6. stolpec v tabeli 4.4, ki vsebujeta rezultate*

izvajanja na CPE]. Moreover, this difference is very consistent: when using OpenCL 1.2 sycl-gtx is about 4% faster, while when using OpenCL 2.0 ComputeCpp is about 5% faster. We do not know the reason for this discrepancy. While we did modify the code from the Codeplay blog to fit the tester and use as much common code as possible, there is no known reason for this almost-exactly-factor-of-two difference. After all, our implementation of SYCL is defnitely less mature. We presume we've made a mistake somewhere, though careful examination of the code did not reveal it. We compared the image outputs, but they were comparable at the same sample" [22].

Tudi mi nismo našli vira teh razlik - naša testiranja pa so prinesla rezultate bolj po vzorcu meritev s ComputeCpp. Ponovitev prvotnih testov pa z naše strani ni mogoča, saj bi potrebovali identično strojno in programsko opremo kot prvotni avtor.

Poglavje 5

Zaključek

V okviru diplomske naloge smo obstoječo implementacijo standarda SYCL - sycl-gtx uspešno prenesli na več računalniških sistemov, na katerih teče operacijski sistem Linux. Prav tako smo implementacijo pognali na specializiranih napravah, ki so na voljo na fakulteti in ki jih prvotni avtor ni uporabil. To so koprocesor Intel Xeon Phi, GPE namenjena računanju - Nvidia Tesla K20 ter zmogljiva CPE, ki podpira OpenCL, Intel Xeon. Enake teste smo izvedli tudi na strojni opremi, ki je namenjena bolj navadnim uporabnikom, z namenom primerjave hitrosti izvajanja. Po tem ko smo sycl-gtx uspešno zagnali na prej omenjenem šolskem sistemu, pa nas je zanimalo ali bi lahko izkoristili koprocesorski način izvajanja kode na koprocesorju Intel Xeon Phi. Prav tako nas je zanimalo, ali bi lahko aplikacijo še dodatno paralelizirali z izvajanjem na dveh identičnih končnih zmogljivih grafičnih napravah. V prvem primeru se je izkazalo, da za tako kombinacijo zahtev ni programske podpore s strani proizvajalca, saj za tak način ne obstajajo potrebne zaglavne datoteke. V drugem primeru, pa bi bilo potrebno kodo „na roke“ razdeliti, za kar se nismo odločili, saj bi s tem zmanjšali prenosljivost obstoječe implementacije.

Ko smo določili, na katerih napravah in na kakšen način lahko uporabljamo implementacijo, smo se na vseh napravah lotili izvajanja enakega testa z namenom primerjave hitrosti izvajanja posamezne naprave in posa-

meznega načina izvajanja. Testni program je uporabljal računsko razmeroma zahteven algoritem za generiranje slike sledenja žarku (angl. *ray tracing*), pri katerem smo z večanjem števila vzorcev na slikovno točko povečevali natančnost končne slike na račun daljšega izvajalnega časa.

Dobljene rezultate smo tudi primerjali z rezultati izvajanja enakega testa s strani avtorja implementacije in ugotovili, da so njegovi rezultati za razred boljši od naših. Naši rezultati so bili bolj podobni rezultatom pridobljenih iz do takrat edine izdane komercialne različice standarda (ComputeCpp). To je potrdilo domnevo, da je pri testiranju prišlo do napake, ki je prvotnemu avtorju prinašala predobre rezultate testov. Pri tem smo tudi končali z nalogo.

Kljub temu, da smo sycl-gtx uspeli pognati na različnih platformah, pa je njegov razvoj še daleč od končanega. Veliko zahtev, ki jih nalaga standard SYCL namreč še ni implementiranih. Kljub temu, pa je to praktično edina odprtokodna implementacija pri kateri razvoj ne stagnira. Prvotni avtor, Peter Žužek, namreč še vedno popravlja in dodaja funkcionalnosti, od njega je tudi odvisna nadaljnja prihodnost razvoja implementacije sycl-gtx.

Literatura

[1] Apple inc. Dosegljivo: https://www.apple.com/. [Dostopano: 23. 8. 2017].

[2] CMake. Dosegljivo: http://www.nvidia.com/object/tesla-servers.html. [Dostopano: 23. 8. 2017].

[3] Codeplay Software, ComputeCpp. Dosegljivo: https://www.codeplay.com/products/computesuite/computecpp. [Dostopano: 25. 8. 2017].

[4] CPE AMD Phenom II X4 955. Dosegljivo: http://www.amd.com/en-us/products/processors/desktop/phenom-ii. [Dostopano: 26. 8. 2017].

[5] CPE Intel Core i7-5600U. Dosegljivo: https://ark.intel.com/products/85215/Intel-Core-i7-5600U-Processor-4M-Cache-up-to-3_20-GHz. [Dostopano: 26. 8. 2017].

[6] Intel Xeon E5-2620. Dosegljivo: http://ark.intel.com/products/64594/Intel-Xeon-Processor-E5-2620-15M-Cache-2_00-GHz-7_20-GTs-Intel-QPI. [Dostopano: 25. 8. 2017].

[7] Koprocesor Intel Xeon Phi. Dosegljivo: https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html. [Dostopano: 24. 8. 2017].

[8] nVidia CUDA. Dosegljivo: http://www.nvidia.com/object/cuda_home_new.html. [Dostopano: 28. 8. 2017].

[9] nVidia Tesla. Dosegljivo: http://www.nvidia.com/object/tesla-servers.html. [Dostopano: 23. 8. 2017].

[10] OpenCL. Dosegljivo: https://www.khronos.org/opencl/. [Dostopano: 23. 8. 2017].

[11] OpenMP. Dosegljivo: http://openmp.org/. [Dostopano: 23. 8. 2017].

[12] SCYL. Dosegljivo: https://www.khronos.org/sycl. [Dostopano: 23. 8. 2017].

[13] Skupina Khronos. Dosegljivo: https://www.khronos.org/. [Dostopano: 23. 8. 2017].

[14] Specifikacija Intel HD Graphics 4600. Dosegljivo: http://www.game-debate.com/hardware/index.php?gid=1438&graphics=Intel%20HD%20Graphics%204600%20Desktop. [Dostopano: 3. 9. 2017].

[15] triSYCL. Dosegljivo: https://github.com/triSYCL/triSYCL. [Dostopano: 25. 8. 2017].

[16] Khronos OpenCL Working Group, SYCL^{TM} Specification. Dosegljivo: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.pdf, 2015. [Dostopano: 23. 8. 2017].

[17] Gordon Brown. Introduction To SYCL. Dosegljivo: https://www.codeplay.com/portal/introduction-to-sycl, 2014. [Dostopano: 30. 8. 2017].

[18] Intel. Koprocesor Intel Xeon Phi. Dosegljivo: https://cmake.org/. [Dostopano: 26. 8. 2017].

[19] Luke Iwanski. SYCL-ing the 'smallpt' Raytracer. Dosegljivo: https://www.codeplay.com/portal/sycl-ing-the-smallpt-raytracer, 2015. [Dostopano: 27. 8. 2017].

[20] nVidia. Specifikacija nVidia Tesla K20. Dosegljivo: https://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf. [Dostopano: 3. 9. 2017].

[21] Peter Žužek. sycl-gtx. Dosegljivo: https://github.com/ProGTX/sycl-gtx. [Dostopano: 23. 8. 2017].

[22] Peter Žužek. Implementacija knjižnice sycl za heterogeno računanje. Magistrska naloga, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, 2016.